The Tacoma Narrows Bridge is a pair of mile-long (1600 meter) suspension bridges with main spans of 2800 feet (850 m), they carry Washington State Route 16 across the Tacoma Narrows of Puget Sound between Tacoma and the Kitsap Peninsula, USA. The first bridge, nicknamed Galloping Gertie, was opened to traffic on July 1, 1940, and became famous four months later for a dramatic wind-induced structural collapse that was caught on color motion picture film.

Learning from Mistakes

- Key idea: Learning from earlier mistakes to prevent them from happening again
- Key technique: Simulate earlier mistakes and see whether the resulting defects are found
- Known as fault-based testing or mutation testing
Seeding Defects

- We seed defects into the program
  generating a mutant — a mutation of the original program
- We run the test suite to see whether it detects the defects (“kills the mutants”)
- A mutant not killed indicates a weakness of the test suite
  The mutant may also be 100% equivalent to the original program
Hans-Peter is moving into this building – actually, he built it, too. He’s worried that everything might be okay. But he’s not that worried.

If you’re building not a building, but a piece of software, you have many more reasons to be worried.
It’s not like this is the ultimate horror…

And this is the summary of structural testing techniques.
Weyuker’s Hypothesis

The adequacy of a coverage criterion can only be intuitively defined.

Established by a number of studies done by E. Weyuker at AT&T. “Any explicit relationship between coverage and error detection would mean that we have a fixed distribution of errors over all statements and paths, which is clearly not the case”.

AspectJ Defect Density

A Bad Test

class TrueStoryTest {
    public int test_all(Object other) {
        executeForSomeTime();
        assertTrue(true);
    }
}

100% coverage – and never fails
Mutation Testing
DeMillo, Lipton, Sayward 1978

Program

A Mutation

class Checker {
    public int compareTo(Object other) {
        return 1;
    }
}

not found by AspectJ test suite

Mutation Operators

<table>
<thead>
<tr>
<th>id</th>
<th>operator</th>
<th>description</th>
<th>constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>cm</td>
<td>constant for constant replacement</td>
<td>replace constant C1 with constant C2</td>
<td>C1 \neq C2</td>
</tr>
<tr>
<td>cr</td>
<td>constant for constant replacement</td>
<td>replace constant C1 with array reference A1</td>
<td>C1 \neq A1</td>
</tr>
<tr>
<td>scr</td>
<td>scalar for constant replacement</td>
<td>replace constant C1 with scalar variable X</td>
<td>X \neq C1</td>
</tr>
<tr>
<td>scr</td>
<td>scalar for constant replacement</td>
<td>replace scalar variable X with constant C1</td>
<td>X = C1</td>
</tr>
<tr>
<td>acr</td>
<td>array for constant replacement</td>
<td>replace constant C1 with array reference A1</td>
<td>C1 \neq A1</td>
</tr>
<tr>
<td>scr</td>
<td>scalar for constant replacement</td>
<td>replace scalar variable X with array reference A1</td>
<td>X = A1</td>
</tr>
<tr>
<td>ssr</td>
<td>struct for constant replacement</td>
<td>replace constant C1 with struct field S</td>
<td>S \neq C1</td>
</tr>
<tr>
<td>svr</td>
<td>scalar variable initialization elimination</td>
<td>remove initialization of scalar variable X</td>
<td>X</td>
</tr>
<tr>
<td>csr</td>
<td>constant for scalar variable replacement</td>
<td>replace scalar variable X with a constant C1</td>
<td>C1 \neq X</td>
</tr>
<tr>
<td>csr</td>
<td>constant for scalar variable replacement</td>
<td>replace scalar variable X with an array reference A1</td>
<td>A1 \neq X</td>
</tr>
<tr>
<td>csr</td>
<td>constant for scalar variable replacement</td>
<td>replace scalar variable X with a struct field S</td>
<td>S \neq X</td>
</tr>
<tr>
<td>sar</td>
<td>scalar for array replacement</td>
<td>replace array reference A1 with scalar variable X</td>
<td>X \neq A1</td>
</tr>
<tr>
<td>sar</td>
<td>scalar for array replacement</td>
<td>replace array reference A1 with scalar variable X</td>
<td>X \neq A1</td>
</tr>
<tr>
<td>cnr</td>
<td>comparable array replacement</td>
<td>replace array reference A1 with a comparable array reference</td>
<td>A1 \neq A1</td>
</tr>
<tr>
<td>sar</td>
<td>struct for array reference replacement</td>
<td>replace array reference A1 with a struct field S</td>
<td>S \neq A1</td>
</tr>
</tbody>
</table>

Expression Modifications
abs: absolute value insertion
aor: arithmetic operator replacement
lcr: logical connector replacement
ror: relational operator replacement
uoi: unary operator insertion
cpr: constant for predicate replacement

Statement Modifications
del: statement deletion
sca: switch case replacement
ses: end block shift
vcs: switch case replacement

If one ever needed a proof that testing is a destructive process—here it is

Figure 16.2: A sample set of mutation operators for the C language, with associated constraints to select test cases that distinguish generated mutants from the original program.
Does it work?

- Generated mutants are similar to real faults
  Andrews, Briand, Labiche, ICSE 2005

- Mutation testing is more powerful than statement or branch coverage
  Walsh, PhD thesis, State University of NY at Binghampton, 1985

- Mutation testing is superior to data flow coverage criteria
Efficiency

- Test suite must be re-run for every single mutation
- Expensive

How do we make mutation testing efficient?
Efficiency

- Manipulate byte code directly rather than recompiling every single mutant
- Focus on few mutation operators
  - replace numerical constant $C$ by $C\pm 1$, or 0
  - negate branch condition
  - replace arithmetic operator (+ by –, * by /, etc.)
- Use mutant schemata
  individual mutants are guarded by run-time conditions
- Use coverage data
  only run those tests that actually execute mutated code

A Mutation

class Checker {
    public int compareTo(Object other) {
        return 1;
    }
}

not found by AspectJ test suite
because it is not executed

Efficiency

- 6.5 CPU hours for Jaxen XPath engine with 12,500 LOC
- Mutation testing is feasible in practice
Inspection

- A mutation may leave program semantics unchanged
- These equivalent mutants must be determined manually
- This task is tedious.

An Equivalent Mutant

```java
public int compareTo(Object other) {
    if (!(other instanceof BcelAdvice))
        return 0;
    BcelAdvice o = (BcelAdvice) other;
    if (kind.getPrecedence() != o.kind.getPrecedence()) {
        if (kind.getPrecedence() > o.kind.getPrecedence())
            return +1;
        else
            return -1;
    }
    // More comparisons...
    return +2;
}
```

no impact on AspectJ

Inspection is Costly

- In a Jaxen sample, 40% of non-detected mutants were equivalent
- Assessing a single mutation took us 30 minutes
- 1,933 mutations were not detected
- This ratio grows as the test suite improves and approaches 100% with a perfect test suite
- Such false positives are just worthless
  Using coverage, false positives at least imply dead

1,000 hours, or 10 weeks for a Microsoft programmer.
Frankl’s Observation

We also observed that [...] mutation testing was costly. Even for these small subject programs, the human effort needed to check a large number of mutants for equivalence was almost prohibitive.


Inspection

How do we determine equivalent mutants?

Aiming for Impact
Measuring Impact

• How do we characterize “impact” on program execution?
• Idea: Look for changes in pre- and postconditions
• Use dynamic invariants to learn these

Dynamic Invariants
pioneered by Mike Ernst’s Daikon

At f(), x is odd
At f(), x = 2

Example

public int ex1511(int[] b, int n)
{
    int s = 0;
    int i = 0;
    while (i != n) {
        s = s + b[i];
        i = i + 1;
    }
    return s;
}

• Run with 100 randomly generated arrays of length 7–13
Obtaining Invariants

- Run
- Trace
- Invariant
- Postcondition
  \[ b[] = \text{orig}(b[]) \]
  \[ \text{return} == \sum(b) \]

Impact on Invariants

```
public LazyMethodGen getLazyMethodGen(String name, 
  String signature, boolean allowMissing) {
  for (Iterator i = methodGens.iterator(); i.hasNext();) {
    LazyMethodGen gen = (LazyMethodGen) i.next();
    if (gen.getName().equals(name) &&
        gen.getSignature().equals(signature))
      return gen;
  }
  if (!allowMissing)
    throw new BCException("Class " + this.getName() +
        " does not have a method " + name +
        " with signature " + signature);
  return null;
}
```

Impact on Invariants

- getLazyMethodGen() mutated method
- UnitDeclaration.resolve() post: target field is now zero
- DelegatingOutputStream.write() pre: upper bound of argument changes
- WeaverAdapter.addingTypeMunger() pre: target field is now non-zero
- ReferenceContext.resolve() post: target field is now non-zero
Impact on Invariants

```java
public LazyMethodGen getLazyMethodGen(String name, String signature, boolean allowMissing) {
    for (Iterator i = methodGens.iterator(); i.hasNext();) {
        LazyMethodGen gen = (LazyMethodGen) i.next();
        if (gen.getName().equals(name) &&
            gen.getSignature().equals(signature))
            return gen;
    }
    if (!allowMissing)
        throw new BCException("Class " + this.getName() +
            " does not have a method " + name +
            " with signature " + signature);
    return null;
}
```

impacts 39 invariants in 18 methods
but undetected by AspectJ unit tests

---

J

- **Mutation Testing Framework for Java**
  - 12 man-months of implementation effort
- **Efficient Mutation Testing**
  - Manipulate byte code directly • Focus on few mutation operators • Use mutant schemata • Use coverage data
- **Ranks Mutations by Impact**
  - Checks impact on dynamic invariants • Uses efficient invariant learner and checker

---

Mutation Testing with Javalanche

---

Program
Mutation Testing
with Javalanche

1. Learn invariants from test suite
2. Insert invariant checkers into code
3. Detect impact of mutations
4. Select mutations with the most invariants violated (= the highest impact)

But does it work?

Evaluation

1. Are mutations with impact less likely to be equivalent?
2. Are mutations with impact more likely to be detected?
3. Are mutants with the highest impact most likely to be detected?
Evaluation Subjects

<table>
<thead>
<tr>
<th>Name</th>
<th>Lines of Code</th>
<th>#Tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>AspectJ Core</td>
<td>94.902</td>
<td>321</td>
</tr>
<tr>
<td>Barbecue</td>
<td>4.837</td>
<td>137</td>
</tr>
<tr>
<td>Commons</td>
<td>18.782</td>
<td>1.590</td>
</tr>
<tr>
<td>Jaxen</td>
<td>12.449</td>
<td>680</td>
</tr>
<tr>
<td>Joda-Time</td>
<td>25.861</td>
<td>3.447</td>
</tr>
<tr>
<td>JT opas</td>
<td>2.031</td>
<td>128</td>
</tr>
<tr>
<td>XStream</td>
<td>14.480</td>
<td>838</td>
</tr>
</tbody>
</table>

Mutations

<table>
<thead>
<tr>
<th>Name</th>
<th>#Mutations</th>
<th>%detected</th>
</tr>
</thead>
<tbody>
<tr>
<td>AspectJ Core</td>
<td>47.146</td>
<td>53</td>
</tr>
<tr>
<td>Barbecue</td>
<td>17.178</td>
<td>67</td>
</tr>
<tr>
<td>Commons</td>
<td>15.125</td>
<td>83</td>
</tr>
<tr>
<td>Jaxen</td>
<td>6.712</td>
<td>61</td>
</tr>
<tr>
<td>Joda-Time</td>
<td>13.859</td>
<td>79</td>
</tr>
<tr>
<td>JT opas</td>
<td>1.533</td>
<td>72</td>
</tr>
<tr>
<td>XStream</td>
<td>5.186</td>
<td>92</td>
</tr>
</tbody>
</table>

Performance

- Mutation testing is feasible in practice
  14 CPU hours for AspectJ, 6 CPU hours for XStream
- Learning invariants is very expensive
  22 CPU hours for AspectJ – one-time effort
- Creating checkers is somewhat expensive
  10 CPU hours for AspectJ – one-time effort

% detected means covered mutations
Are mutations with impact less likely to be equivalent?

- Randomly selected non-detected Jaxen mutants – 12 violating, 12 non-violating
- Manual inspection: Are mutations equivalent?
- Mutation was proven non-equivalent iff we could create a detecting test case
- Assessment took 30 minutes per mutation

In our sample, mutants with impact were significantly less likely to be equivalent.

Difference is statistically significant according to Fisher test.
Mutations and tests made public to counter researcher bias.
Are mutations with impact more likely to be detected?

1. Mutations detected by the test suite are non-equivalent.
2. The more of my mutations are detected, the fewer equivalent mutations I have generated.

Mutations with impact are more likely to be detected by actual tests – and thus less likely to be equivalent.
Are mutations with the highest impact most likely to be detected?

Mutations with the highest impact are most likely to be detected by actual tests – and thus the least likely to be equivalent.

Detection Rates

Evaluation

1. Are mutations with impact less likely to be equivalent? ✔
2. Are mutations with impact more likely to be detected? ✔
3. Are mutants with the highest impact most likely to be detected? ✔
Future Work

- How effective is mutation testing on a large scale – compared to traditional coverage
- Alternative impact measures
  Coverage • Program spectra • Method sequences
- Adaptive mutation testing
  Evolve mutations to have the fittest survive
- Automatic fixes
  Choose fixes (mutations) with minimal impact

Factor 6,666 – plus full automation due to lack of inspection
Estimating #Defects

- How many defects remain in our software?
- With mutation testing, we can make an estimate of remaining defects

Let's consider a lake. How many fish are in that lake?

Simple. We catch a number of fish (say, 1000), tag them, and throw them back again.

Fish Tag

- We catch 1,000 fish and tag them
Let’s assume over the next week, we ask fishermen to count the number of tags. We find 300 untagged and 50 tagged fish.

...and we can thus estimate that there are about 6,000 remaining untagged fish in the lake.

That’s how we can tell how many fish there are.
Now let's assume our lake is not a lake, but our program.

A Mutant

- We seed 1,000 mutations into the program

Simple. We catch a number of fish (say, 1000), tag them, and throw them back again.

Our test suite finds 50 mutants, and 300 natural faults.
Estimate

\[
\frac{1,000}{\text{remaining defects}} = \frac{50}{300}
\]

… and we can again estimate that there are about 6,000 remaining defects in our program. (A test suite finding only 50 out of 1,000 mutations is a real bad sign.)

Conclusion

Assumptions

- Mutations are representatives for earlier mistakes
  so-called competent programmer hypothesis

- Failures come to be because of a combination of minor mistakes
  but there may be logical errors that cross-cut the program

- These hypotheses are not proven
http://www.st.cs.uni-saarland.de/mutation/

Mutation Testing with avalanche
1. Learn invariants from test suite
2. Insert invariant checkers into test
3. Detect impact of mutations
4. Select mutations with the most
   invariants violated
   (i.e. the highest impact)